Appendix B. XML Schema Basics

The XML Schema specification is long and complex. To create SOAP and WSDL XML, you must know how XML Schema specify data types. This appendix is a quick introduction to the topic, with examples. You won't come away a Schema guru; you will be able to follow WSDL.

B.1 Simple and Complex Types

In an XML Schema, all data types are either *primitive* or *derived*. A primitive data type is one that cannot be expressed in terms of any other data type. The XML Schema specification gives the example of a *float*, "a well-defined mathematical concept that cannot be defined in terms of other data types," where an *integer* is a derivative of *decimal* data type. In this case, a float is primitive and an integer is derived.

All *primitive* data types are *atomic*. That is, the value of the data type cannot be broken down any more than it already is. For example, the number 1 is an atomic value.*Derived* data types may or may not be atomic. For example, an integer as we have already seen is a derived data type that has an atomic value. A telephone number, however, is also a derived data type whose value is not atomic; it is actually a collection of three individual atomic values.

Data types are mainly derived through *restriction* or *extension* (there are other ways, but these are the most common). In derivation through restriction, the value of the data type is restricted in some way. For example, an integer is a derivation of the decimal data type that allows for a narrower range of values than does a decimal; an integer, in other words, is allowed to contain a restricted subset of decimal values. Derivation through extension means that various restrictions on the base data type are being lifted to allow additional values that otherwise wouldn't be allowed. For example, a telephone number data type may be extended to include a country code field.

This is somewhat analogous to Java classes and objects. All Java classes are types of Java objects. All Java objects are of type `java.lang.Object`. When I create a new Java class that derives from `java.lang.Object`, most of the time I am adding new functionality (a new operation, a new property, etc). This is *derivation by extension*. When I override an existing operation (such as the `toString( )` operation), I am *deriving by restriction*. This analogy obviously doesn't bear close examination, but may be useful nonetheless.

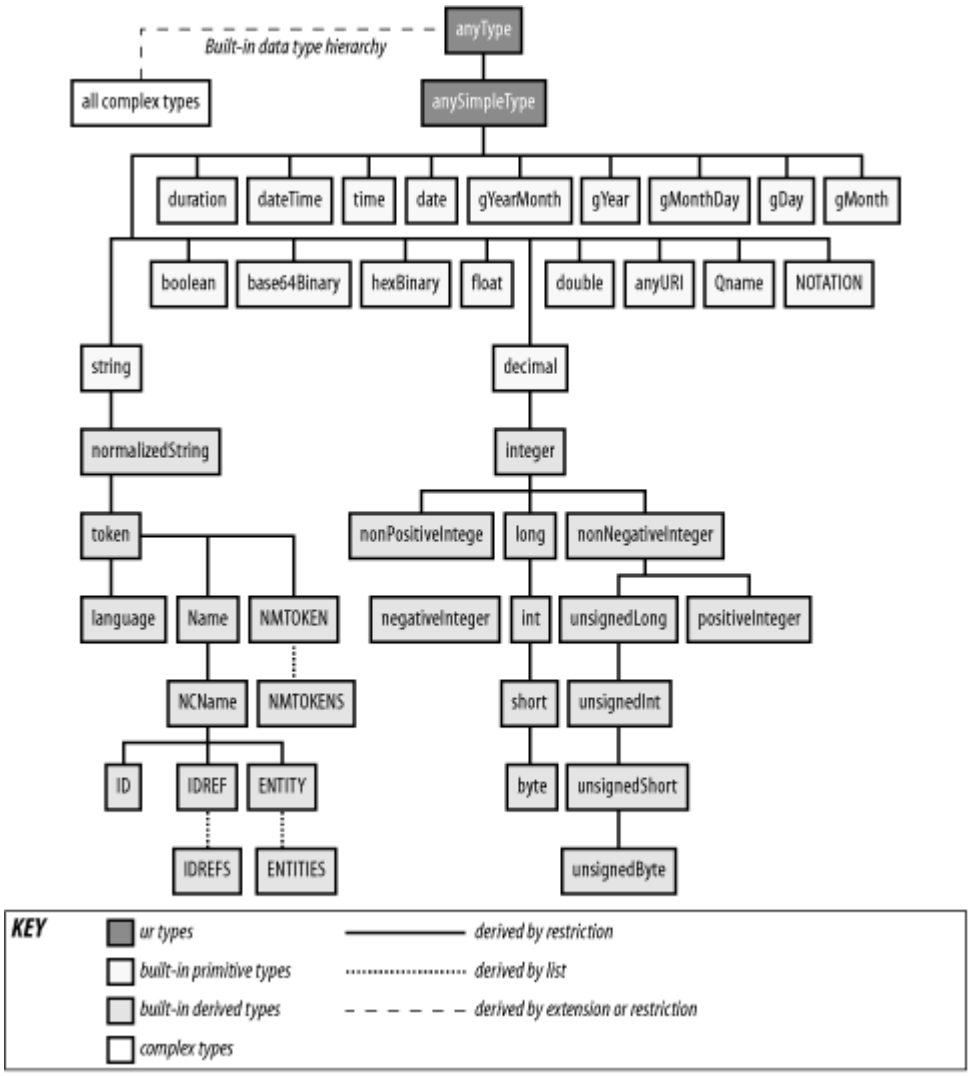
The authors of the XML Schema specification realized that while they had a simple and extensible data typing mechanism, they still needed to define a handful of built-in data types that reflect common use scenarios. That way, application developers wouldn't have to keep reinventing the same common data types time and time again, which would just end with the same confusion that interferes with interoperability between programming platforms. So the built-in XML Schema data types were born and we now have things like `string`, `integer`, `float`, `boolean`, `URI`, and `time` finally defined in a common way that all application platforms are capable of understanding.

These data types form a hierarchy that can be traced back to a single primitive atomic data type called `anyType`. All other data types used in XML Schemas derive from this single primitive type.

There are two kinds of data types that can be derived from `anyType`: simple types and complex types. Simple types represent all derived, atomic data types built into XML Schema. This includes things like `string`, `integer`, and `boolean`. Complex types represent all derived, nonatomic data types—the telephone number, for instance.

Figure B-1, adapted from the one used in the XML Schema data type specification, illustrates the hierarchy of built-in data types.

Figure B-1. Hierarchy of built-in data types in XML Schemas



Be sure to notice that every built-in "simple type" does not derive directly from `anyType`, but from the `anySimpleType` data type, which is itself a derivative of `anyType`. As a rule, the XML Schema specification dictates that any derivative of `anySimpleType` cannot be derived by *extension*. Basically, this means the element cannot contain any attributes or child elements, in terms of expressing the data type as XML. Again, if this isn't making much sense, it will soon as we look at a few simple examples.

A quick review: we introduced the fact that there are essentially two types of data defined by an XML Schema. These include simple types, which are atomic. Single value data types that may or may not be derived through restriction from other simple types. The other type of data defined by an XML Schema is complex types, which are composed of collections of simple types and must be derived either from other complex types or simple types.

B.2 Some Examples

While the XML Schema data typing mechanism is actually quite easy to use, we have found that it is often a difficult thing to explain. Let's walk through some simple examples to clear things up.

B.2.1 Simple Types

Let's practice defining a simple data type. Say we have a `productCode` data type. This product code must start with two numbers followed by a dash and five more numbers. Example B-1 illustrates how to express this data type within an XML Schema.

Example B-1. productCode

```
<xsd:simpleType name="productCode">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{2}-\d{5}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Here, we see that `productCode` is a derivative of the XML Schema built-in data type `string` that has been restricted to only allow values that match the regular expression `\d{2}-\d{5}`. If we were to express an instance of this data type in XML, it would look something like Example B-2.

Example B-2. Instance of productCode

```
<pCode xsi:type="abc:productCode">12-12345</pCode>
```

In this simple example, we demonstrate several things: the `productCode` is a derived simple type with an atomic value, and we derive the `productCode` by restricting the possible set of values that its base data type (in this case `string`) can contain.

Now let's create an extended product code that may or may not have an additional `-[a-z]` (a dash followed by any lowercase letter). We could do this by deriving a new `productCodeEx simpleType` and changing the pattern to `\d{2}-\d{5}(-[a-z]){0,1}`, as in Example B-3.

Example B-3. Extended productCode

```
<xsd:simpleType name="productCodeEx">
  <xsd:restriction base="productCode">
    <xsd:pattern value="\d{2}-\d{5}(-[a-z]){0,1}"/>
  </xsd:restriction>
</xsd:simpleType>
```

B.2.2 Complex Types

Now you've probably got the hang of simple types and are itching to look at complex types. A complex type is any data type that contains a collection of other primitive data types. A telephone number is an example. It contains three distinct pieces of information. A telephone number complex type in XML Schema looks like Example B-4.

Example B-4. telephoneNumber type

```
<xsd:complexType name="telephoneNumber">
  <xsd:sequence>
    <xsd:element name="area">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="\d{3}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="exchange">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="\d{3}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="number">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="\d{4}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The `telephoneNumber` data type consists of a sequence of three data elements, each of which are restricted derivatives of the XML Schema `string` data type. An instance of this data type would look something like Example B-5.

Example B-5. Instance of telephoneNumber

```
<telephone xsi:type="abc:telephoneNumber">
   <area>123</area>
   <exchange>123</exchange>
   <number>1234</number>
</telephone>
```

If I were to go back and create an extended version of this data type that includes a country code, I would do so by creating a new `complexType` derived by extension. This is shown in Example B-6.

Example B-6. Extending telephoneNumber to include a country code

```
<xsd:complexType name="telephoneNumberEx">
  <xsd:complexContent>
    <xsd:extension base="telephoneNumber">
      <xsd:sequence>
        <xsd:element name="countryCode">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="\d{2}"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

An instance of the extended telephone number would look like Example B-7.

Example B-7. Instance of extended telephoneNumber

```
<telephone xsi:type="abc:telephoneNumber">
   <area>123</area>
   <exchange>123</exchange>
   <number>1234</number>
   <countryCode>01</countryCode>
</telephone>
```

Notice that the `countryCode` element is at the end of the sequence of data elements. This is due to the way that XML Schema enforces element ordering within data types. Because we are deriving by extension, all new elements defined in the `telephoneNumberEx` data type have to appear after the elements defined in its base `telephoneNumber` data type. If we wanted `countryCode` to appear first in the sequence, we would actually have to derive by restriction and redeclare each of the data elements, as in Example B-8.

Example B-8. restricted telephoneNumber

```
<xsd:complexType name="telephoneNumberEx">
    <xsd:complexContent>
      <xsd:restriction base="telephoneNumber">
        <xsd:sequence>
          <xsd:element name="countryCode">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:pattern value="\d{2}"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="area">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:pattern value="\d{3}"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
```

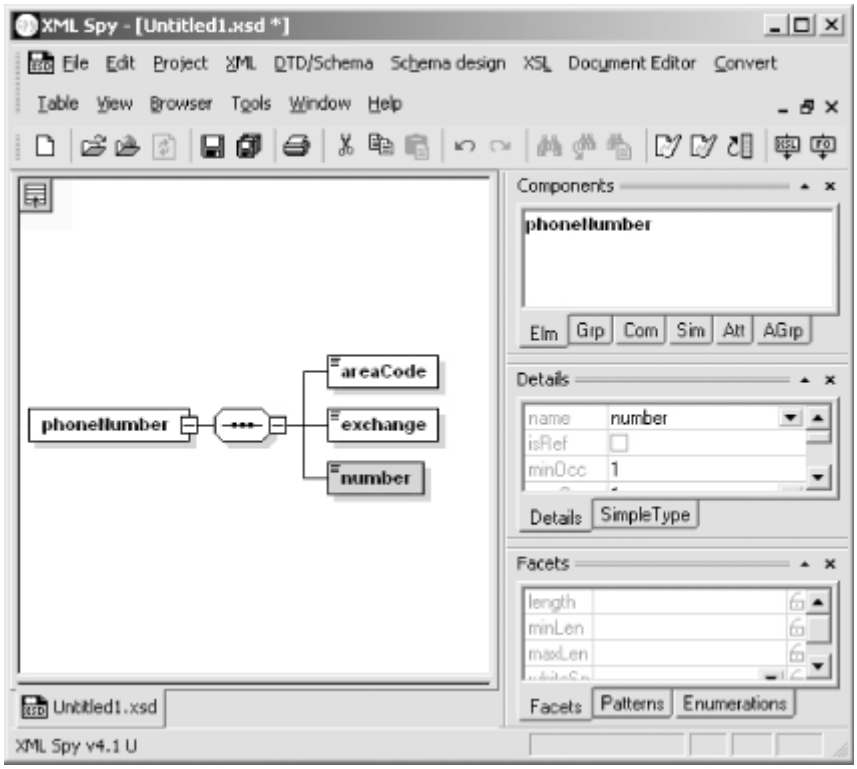
```
        <xsd:element name="exchange">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="\d{3}"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="number">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="\d{4}"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

And that's the basics of defining data types with XML Schemas. There are plenty of details that we are leaving out. It's worthwhile taking the time to learn more about XML Schemas.

B.3 XML Spy

XML Spy is perhaps the best product available for working with XML Schemas. Its XML development environment allows you to visually design XML Schemas quickly and easily, hiding away the syntactic complexity that normally trips people up. Figure B-2 shows a screenshot of XML Spy's visual schema editor.

Figure B-2. A view of the XML Spy visual schema editor



XML Spy is a commercial product available from http://www.xmlspy.com/. Though it's not cheap (a few hundred U.S. dollars at the time of this writing), it is well worth the price for serious developers.